NASA-CR-193051

PROCEDURAL REASONING SYSTEM USER GUIDE

Amy L. Lansky
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025

July 19, 1985

(NASA-CR-193051) PROCEDURAL REASONING SYSTEM USER GUIDE (SRI International Corp.) 38 p

N93-72546

Unclas

Z9/61 0163065

Contents

This document is designed to introduce users to SRI International's Procedural Reasoning System (PRS)[1] on the Symbolics 3600. The user should have some rudimentary knowledge of the Lisp Machine, but even without it, should be able to muddle through with the directions given below and some experimentation. The user should also already be familiar with the concepts behind procedural logic (see the IJCAI-85 paper, "A PROCEDURAL LOGIC," by Georgeff, Lansky, and Bessiere). Don't be afraid to ask someone for help![2]

1 General Overview

The Procedural Reasoning System (PRS) is a framework for describing procedural forms of knowledge, and executing those procedures in a very flexible, reactive way. By procedural knowledge we mean knowledge about the consequences of performing actions in a specific way. For example, we might know that a sequence of actions of form - "*put clothes in washer,*" "*put soap in washer,*" "*turn washer on,*" "*wait 45 minutes*" - will achieve a goal of form "*clean the clothes.*" Much of people's knowledge about the everyday world is encoded in procedures such as this one - perhaps *most* of our knowledge.

In order to use the PRS system, a user specifies domain knowledge in either procedural form or in terms of facts or beliefs placed in an updatable, nonmonotonic data base. Those pieces of knowledge that are specifically procedural in nature are sometimes referred to *Knowledge Areas (KAs)* or *processes*. Each piece of procedural knowledge is represented within the system as a graphical network that encodes the steps of the procedure. A procedure must also be associated with information stating under what situations it may be used, as well as what it is useful for (i.e., a declaration of what types of goals the procedure can be used to achieve, and under what situations it is truly applicable). The user of the PRS system inputs all of this procedural information via a graphical network editor called GRASPER II.

A typical example of a graphical process network is given in figure 1. It describes a procedure to align all of the wheels on a car. Each arc of the network is labeled with a goal. Execution is meant to begin at the **START** node in the network, and proceeds by following arcs through the network. Execution completes, achieving the *purpose* or *goal* of the entire procedure, if execution reaches a finish node - a node with no exiting arcs. If more than one arc emanates from a given node, any of the arcs emanating from that node may be transitted - only one of them need be transitted successfully. To transit an arc, the system must find a procedure that achieves the goal labelling that

[1] Also known as *Peritus II.*

[2] Throughout this document, you will notice references to the acronym *PES* (Procedural Expert System). This should be considered equivalent to PRS - it is the old name for the system.

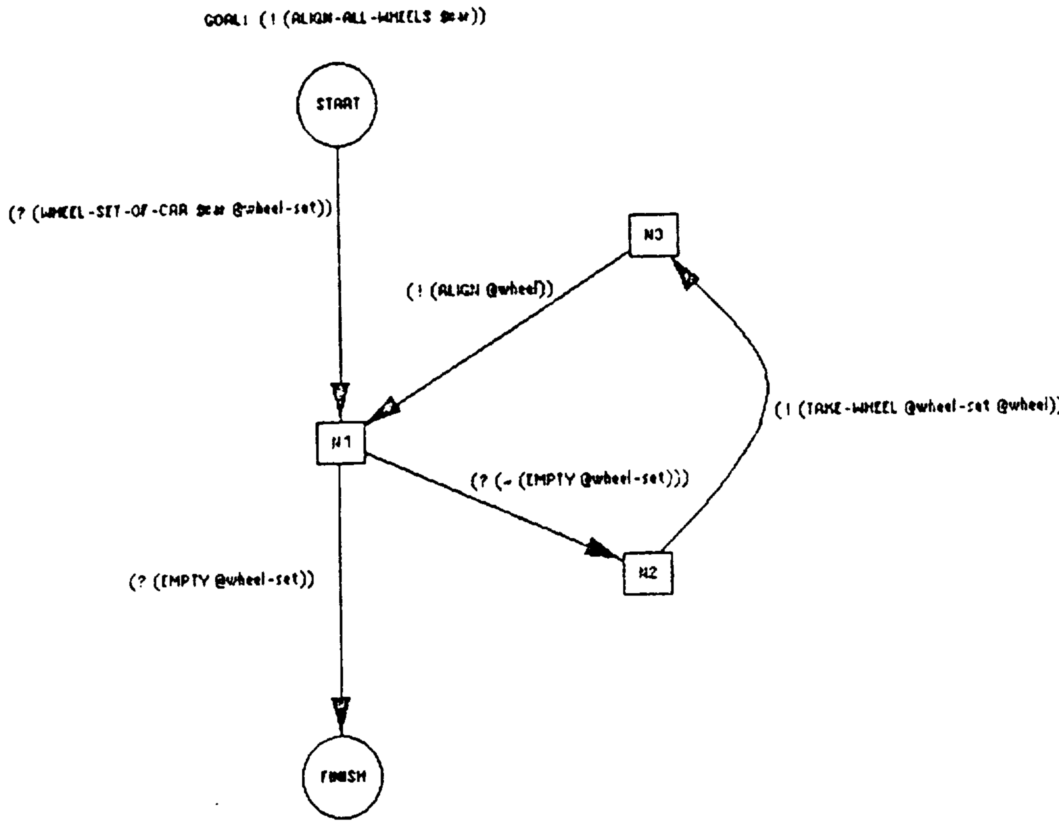


Figure 1: A Simple Process Network

arc. If the system fails to transit an arc (which emanates from a node, say, n), then other arcs emanating from n may be tried. If, however, the system fails to achieve any of the goals on arcs emanating from n, the procedure as a whole will fail. For example, since only one arc emanates from node $N3$ in figure 1, if an attempt to align any given wheel fails, the alignment procedure for the entire car will have failed. The exact notation for arc labels and variable usage will be described later in this document.

As stated above, a user's domain description may also include a data base of facts or "beliefs." Another type of information encoded within the PRS system is a set of goals to be achieved. A complete domain description might thus consist of, say, a set of process networks that describe procedures used for trouble-shooting a complex piece of equipment, a data base that describes the structure of the equipment as well as current failure indications, domain goals that seek the determination of a faulty module.

From a conceptual viewpoint, the PRS system operates in a relatively simple way. At any particular point in time, certain goals are active in the system, and certain facts or beliefs are held in the database. Given these extant goals and facts, a subset of procedures (processes) in the system will be relevant. One of these processes will then be chosen to execute. In the course of process network execution, new goals will be formulated and new facts and beliefs may be derived and inserted into the data base. At such points, new relevant processes are once again found and possibly invoked.

One point that the reader may have already noticed is that PRS *reactive*, rather than merely goal-driven. That is, processes may respond not only to goals, but also to facts. For example, when a new fact enters the data base, the execution of the currently active process network might be suspended, with a new relevant process network taking over. One of the ways the system resolves which process to execute at any given time is by using other *meta-level* process networks to help make such decisions. These meta-level processes are manipulated and invoked by the system in the same way as any other process. However, they respond to facts and goals of the system itself, rather than just those of the application domain. Meta-level processes are described in more detail in section 4.1 and in appendix D.

1.1 System Structure

It may be useful for the user to view the system as consisting of four logical components:

- **The Data Base.** The system data base stores all current beliefs within the system. Part of the information supplied by a user of the PRS system is an initial set of facts (beliefs) that should be put into this data base.

- **The Goal Stack.** The goal stack stores all current goals that are unresolved in the system.

- **Processes.** Each PRS system is associated with a set of processes which describe how to achieve particular goals, as well as how to infer and conclude specific facts into the data base. Some of these processes may be relevant to a particular domain. Others may be *meta-level* processes – they contain information about the manipulation of the PRS system itself (for example, how to choose between multiple relevant processes, or how to achieve a conjunction, disjunction, or the negation of goals). Each PRS system contains processes coming from two sources: those that are supplied by you, the user, (these constitute the application "program"), and those processes that are a default part of every PRS system.

- **The Interpreter.** The system interpreter runs the entire system. When new goals are pushed on the goal stack, the interpreter checks to see if any new processes are relevant, and executes them. Likewise, whenever a new fact is concluded to the database, the interpreter will perform appropriate truth maintenance functions and possibly activate new applicable processes.

1.2 User Files

Each user application system potentially consists of three files, which provide various information to the different parts of the system described above. The first file describes the set of user-defined application processes, the second contains a set of initial data base facts, and the third is a file of user-defined functions that may be relevant to the particular application. The contents of these three files, and how to build them, will be described in sections 4, 5, and 6. We begin with the short description below.[3]

- `YOUR-FILESERVER:>USERNAME>YOUR-APPLICATION-PROCESSES.lisp`
  The *processes* file is the heart of any user application and contains a set of processes constructed using the GRASPER system. Section 4 describes exactly how to set up a processes file.

- `YOUR-FILESERVER:>USERNAME>YOUR-APPLICATION-DATA-BASE.lisp`
  The *database* file is an optional part of a user application. It contains an initial set of facts that must loaded into the system database before the application can be run.

- `YOUR-FILESERVER:>USERNAME>YOUR-APPLICATION-FUNCTIONS.lisp`
  The *functions* file is also an optional part of a user application. It contains any user-defined functions that are needed or useful for running an application. These will typically include application-specific *actions* (see section 4.1), and user-defined functions that aid you in running the application system. For example, they may be macro operations for concluding or deleting specific facts from the database, or pushing specific goals on the goal stack.

[3]Throughout this document, capital letters are used for tokens that should be replace by user-specific tokens. For example, `YOUR-FILESERVER:>USERNAME>YOUR-APPLICATION-PROCESSES.lisp` might be replaced by `ritter:>lansky>foo.lisp`.

2 Getting Started

Each user of the Lisp machine must have an account and file directory. See a Lisp machine wizard to have one set up (at SRI, Mabry Tyson or Paul Martin). Once your directory has been set up, log in by typing `(login 'USERNAME)`.[4] Notice that the bottom banner of the Lisp Listener window includes a location that says USER:. This is where the the current *package name* is given. Right now you are in the USER package.[5].

Now you are ready to load the PRS system. Type to the Lisp Listener window

```
(make-system "pes")
```

You will find that the system takes quite a long time to load up - perhaps 5 minutes. The first thing loaded will be the GRASPER system, then the basic PRS system itself. To transfer to the PRS window, you must now type SELECT-A. Whenever you transfer to the PRS window in this fashion, the package notation in the bottom banner should change to read PES:.[6] In general, you should always make sure you are in the PES package when using this system. Otherwise, unexpected forms of behavior may occur.

First time users of the system may at this point want to try out a run of the PRS system on a demonstration application. Section 7 describes how to run the RCS system, an application system that does fault-diagnosis for a portion of the Reaction Control System of NASA's space shuttle.

[4]In general, this document uses capital letters to denote an object that is replaced with some user-specific token. In this case, for example, you might type `(login 'lansky)`.

[5]If this is not the case, type to the Lisp Listener window, `(PKG-GOTO 'USER)`. For a more full description of packages on the Lisp Machine, see appendix B

[6]If this is not the case, type `(PKG-GOTO 'PES)` to the PRS window.

3 The Menu System

The PRS menu system is the interface through which a user creates a new application system, loads an existing system, makes changes to a system, or runs a system. The top level PRS menu is presented to you whenever you click once on any of the mouse buttons in a PRS window. To leave this or any other menu, just move the mouse cursor out of the menu region.

The top level menu contains the following commands (lower level menus will be described in the next section):

- **LOAD**
  Guides you through loading an application that has already been set up. In order to load an application, you must know the names of its component files. Usually there are three such files, the *processes* file, the *database* file, and the *functions* file. The way these files are created is described later in this document. Note that it is also possible to load processes into the system directly from GRASPER. In addition, some applications may begin with an empty database, or may require no user-defined functions. Thus, none of these application files are strictly necessary.

- **EDIT**
  Serves as an interface for creating and editing any of the user application files.

- **RUN**
  Helps you run a loaded application system. A lower level menu is provided for concluding facts into the system database, or putting new goals on the goal stack, and is thus a vehicle for getting system processes to respond and execute. Ultimately, `RUN` will also be the interface for deleting facts, or evaluating user defined functions.

- **SAVE**
  Enables you to save any processes file, database file, or functions file to the file system.

- **TRACE**
  Enables you to turn tracing of process execution on and off. Right now, tracing of process execution is purely textual. Eventually, processes will be traced graphically in a special run-time interaction window.

- **HELP**
  Prints documentation of these commands.

As a rule, usage of the PRS system will normally follow the following pattern:

- *Creation of Application System:* Use EDIT.
- *Testing of Application System:* Repeated use of the following cycle:
  1. LOAD to load the system.
  2. RUN to run the system.
  3. EDIT to modify the system.

3.1 Lower Level Menus

In general, the PRS menu system should be self explanatory. However, a brief rundown of the various lower level menus is in order, before we describe how to build the user application files.

3.1.1 EDIT

If you click EDIT, you will be given the following menu of choices of how to create and manipulate user application files:

```
CURRENT PROCESSES -- view current processes in GRASPER
CREATE  PROCESSES -- create a new set of processes in GRASPER
APPEND  PROCESSES -- merge a processes file into those in GRASPER
REPLACE PROCESSES -- place a new file of processes into GRASPER
DATABASE          -- edit a file of data base facts
FUNCTIONS         -- edit a file of lisp functions
HELP
```

The first four choices deal with the creation and manipulation of processes files. All four switch you into the GRASPER context. When you wish to exit GRASPER and return to the PRS window, just type SELECT-A, as usual. The use of GRASPER itself, and instructions for how to create process descriptions will be discussed in section 4.

- CURRENT PROCESSES :
  By clicking on this choice, you will simply be switched into GRASPER and be given the opportunity to view and/or edit whatever processes are currently loaded there.
- CREATE PROCESSES :
  By clicking this choice, you will be given the opportunity to create a fresh processes file in GRASPER. Any processes that were previously in GRASPER will be overwritten (a warning message will tell you this). First you will be asked the name of your intended processes file. If that file already exists, a warning message will ask you if you just want to load that file, specify a new name, or abort.

Then you will be placed into GRASPER. At this point you may create or modify processes as you wish. If you wish to save the current GRASPER context to the file system, you may do so through the GRASPER menu options (see section 4) or through the **SAVE** menu described below. In any case, GRASPER will always retain its current set of processes, from one use of GRASPER to the next, unless those processes are explicitly overwritten.

- **APPEND PROCESSES** :
  This choice enables you to append or *merge* an already existing set of processes specified in some file, in with those processes currently loaded into GRASPER. Thus, you can build a set of processes not only by entering new process networks within GRASPER, but also by merging several already constructed processes files. First, you will be asked for the name of the file containing the processes to be merged into GRASPER. If that file does not exist, you will be asked whether you just want to proceed and enter GRASPER anyway, specify a new file name, or abort. Next you will put into GRASPER, which now contains the new processes you have just added, merged in with the old ones.

  Notice that this operation is a true merge rather than a add/replace. Thus, if a new process named P is merged in, and one named P already existed, some unexpected things might happen. If the old and new versions of P are similar, then they will be truly merged - i.e. the new resulting P will have the union of edges and nodes of the two together, and edge names will be unioned as well. If the two versions are completely different, GRASPER will probably bomb out. Given this semantics for **APPEND**, if you really want to replace an old process P with a new version in some file, it is best to delete the old version of P first, before executing the append operation.

- **REPLACE PROCESSES** :
  This choice enables you to totally replace the current contents of GRASPER with an entirely new set of processes in some specified file. Any processes that were previously in GRASPER will be overwritten (a warning message will tell you this). First, you will be asked for the name of the file containing the processes to be loaded into GRASPER. If that file does not exist, you will be asked whether you just want to proceed and enter GRASPER anyway, specify a new file name, or abort. Next you will put into GRASPER, which now contains the new processes you have just loaded.

The final two choices in the **EDIT** menu, **DATABASE** and **FUNCTIONS**, enable you to create or edit a database file and a functions file, respectively. Both operate in basically the same way. If you click **DATABASE** or **FUNCTIONS**, you will be asked for the name of the database or functions file you wish to create or edit. The PRS system will then automatically switch to the ZMACS editor context for you, and load the specified file into the editor window. To return to the PRS window when you are through, type SELECT-A as usual.

3.1.2 LOAD

The LOAD menu is used to load the processes, data base, and functions of an application into the PRS system itself. The files containing this information should have already been set up using the EDIT menu as described above.

After clicking LOAD, you will be given the loading menu with the following commands:

```
INITIALIZE PROCESSES -- reinitialize the set of system processes
APPEND PROCESSES     -- add some new processes into the system
INITIALIZE DATABASE  -- reinitialize the data base with new facts
APPEND DATABASE      -- add some facts from a file into the data base
LISP FUNCTIONS       -- load a file of lisp functions
HELP
```

The first two commands help you load any processes files you may have built into the PRS system. By *load* we mean, load each of the process networks, compile those networks into an executable form, and perform an analysis on the invocation part of each process to determine and record those situations under which that process is applicable. Note that just because processes are loaded into GRASPER, does not mean those processes are loaded in the PRS system. In a way, GRASPER may be viewed as an editor, and the loading process may be viewed as compiling a program for execution.

Normally, a user will click INITIALIZE PROCESSES to load a processes file. If you click INITIALIZE PROCESSES you will be completely reinitializing the system's internal set of processes - that is, any previously loaded processes will be discarded. In contrast, APPEND PROCESSES has the same functionality as INITIALIZE PROCESSES (it loads a processes file), except it *adds* those processes to those that may have already been loaded. After clicking INITIALIZE PROCESSES or APPEND PROCESSES, you will be given a menu with two items, FILE and GRASPER. If you click on FILE, you will be asked for the name of the processes file you wish to load. That file will then be loaded both into GRASPER as well as the PRS system itself. If you click on GRASPER, those processes which are currently loaded into GRASPER will be loaded into the PRS system.

To load/initialize the PRS data base, the user will click on the commands INITIALIZE DATABASE and/or APPEND DATABASE. INITIALIZE DATABASE will completely reinitialize the data base - all previously loaded facts will be discarded. Then the user will be given a menu with two options, FILE or NULL DATABASE. If you click FILE, the system will ask for the name of a file of data base facts to load, and will then proceed to load them into the data base. If you click NULL DATABASE, the data base will be left in an empty state. In contrast, APPEND DATABASE has the function of simply adding a file of facts to the current system data base. The user is asked for the name of a file of data base facts, and that file is subsequently loaded into the data base.

Finally, to load a file of user-defined lisp functions, one would click on the command LISP FUNCTIONS. You will be asked for the name of a file to load, and the system will then load that file.

3.1.3 RUN

If you click RUN, you will be given a menu with commands FACT, GOAL, HELP. The FACT and GOAL commands enable you to conclude facts to the database, and push goals onto the system goal stack, respectively. By *concluding* a fact we mean adding that fact to the PRS system data base and checking to see if it makes any processes relevant. By concluding facts or pushing goals onto the stack, the user can get the system running in response to the newly concluded fact or pushed goal. This menu will be repeatedly given to you so that you can conclude new facts or push new goals.

Alternatively, you may wish to get your application system running by invoking your own user-defined functions. These functions can be invoked directly in the PRS window. Typically, such functions will be defined by you in such a way that they conclude certain facts to the database, push goals onto the goal stack, etc. The purpose of defining your own functions is merely as a convenience - they serve as macro operations, so that you don't have to manually add a long set of facts and goals to the system every time you wish your system to run. Section 6 discusses how to write these functions. Ultimately, the RUN menu will contain commands for deleting facts, evaluating functions, or invoking other commands that might be useful for running an application system.

3.1.4 SAVE

If you click on SAVE you will be given a menu with the commands PROCESSES, DATABASE, FUNCTIONS. If you click on PROCESSES, the set of processes currently in GRASPER will be saved to the file that you specify. Similarly, if you click on DATABASE or FUNCTIONS, the file that you have designated as your data base file or functions file within the editor buffers will be saved to the file system.[7]

[7]These last two operations (DATABASE and FUNCTIONS) have not yet been implemented.

3.1.5 TRACE

If you click TRACE you will be given a menu with commands NO-TRACING, TRACE-ALL-PROCESSES, TRACE-SOME-PROCESSES. NO-TRACING turns off all tracing of process execution. TRACE-ALL-PROCESSES turns back on tracing of all system processes. TRACE-SOME-PROCESSES enables you to specify a subset of processes to be traced. The system will, by default, trace *all* processes unless you otherwise specify via this TRACE menu.

4 Processes File: `YOUR-APPLICATION-PROCESSES.lisp`

Each PRS system must include a set of processes supplied by the user. Normally, process descriptions are stored within a "processes" file. Each process description includes a network of labeled nodes and edges, as well as an invocation condition that describes the situations in which that process is applicable and useful.

The system for inputting these process descriptions is GRASPER, a multi-purpose program for creating networks of labeled nodes and edges, designed by John Lowrance and Tom Strat. After constructing your process descriptions in GRASPER, you can use the GRASPER output command (click GRAPH, then click OUTPUT) to store them into the file of your choice. The **SAVE** menu can also be used to store the set of processes currently in GRASPER to a file.

Being a general multi-purpose system, GRASPER has its own set of terminology for referring to processes networks.[8] In GRASPER "lingo," a user file is called a *graph* and consists of a set of *spaces*. For our purposes, each GRASPER space will be a process description. Thus, your entire "processes" file is really a GRASPER "graph" that consists of a set of processes descriptions (or GRASPER spaces).

The GRASPER user interface is fairly straightforward, so you should just try experimenting with it.[9] There are two menus, the one on the left for choosing what type of object you wish to deal with (an entire graph, a space, a node, or an edge), and the one on the right for choosing which operation you'd like to perform on that object.

As stated above, each process description (or space) consists of a directed network of labeled nodes and edges, as well as an invocation condition. To create a new process description, proceed as follows. First, tell GRASPER to create a space (click SPACE, then CREATE). It will ask you for a name for that space. The space may then be associated with a VALUE. This is where you will store the invocation condition of a process. To type in the invocation condition, you must edit the value of the space. Click SPACE, then EDIT VALUE, and then type in the invocation part for the process you are creating.[10] The appropriate syntax for the invocation part of a process is as follows:

---

[8] Eventually, we plan on building our own GRASPER-based facility whose interface will use terminology more consistent with our own.

[9] If you get stuck in GRASPER, try either SELECT G or ABORT or c-m-ABORT followed by (**GRASPER:gg**). In general, SELECT G will always bring you to the GRASPER window.

[10] GRASPER actually represents the VALUE of a space as a LISP association list. Thus, by typing in the invocation part you are telling GRASPER that your space is associated with a key-value pair with key-name **INVOCATION-PART**.

```
<value of a process space> ::= ((INVOCATION-PART <invocation condition>))

<invocation condition> ::= <goal>
                       ::= <fact>
                       ::= <lisp predicate>
                       ::= (AND <invocation condition> ...
                                <invocation condition>)
                       ::= (OR  <invocation condition> ...
                                <invocation condition>)

<goal> ::= (*goal (<goal-sym> <literal>))

<fact> ::= (*fact <literal>)

<lisp predicate> ::= (*lisp-predicate <any lisp predicate>)

<goal-sym> ::= !
           ::= ?
           ::= *

<literal> ::= (<predicate-name> <param> ... <param>)
          ::= (~ (<predicate-name> <param> ... <param>))

<predicate-name> ::= <string>

<param> ::= <string>
        ::= <var>

<var> ::= $<string>
      ::= @<string>
      ::= %<string>
```

Note that a "string" parameter may be an atom, or even a lisp function call (which may use variables). PRS variables come in three flavors: *global* variables, which are prefixed by **\$**; *program* variables, which are prefixed by **@**; and *local* variables, which are prefixed by **%**. For an explanation of variable usage, see Appendix C. In general, only global variables are used in an invocation part, whereas all types of variables may be used in the goal expressions labelling edges (see below).

As an example of an invocation condition, we might have the following:

```
((INVOCATION-PART (AND (*goal (! (~ (P $x $y))))
                       (*fact (G $x baz))
                       (*lisp-predicate (LISTP $x)) )))
```

It states that the particular process being specified is applicable precisely when a goal of form (! (~ (P $x $y))) is on the goal stack, *and* the fact (G $x baz) is in the database, *and* the evaluation of the lisp predicate (LISTP $x) yields T (true).

Once you have typed in the invocation part of the process and made it the value of the space, you may go on and create the process network. Each node must have a unique name, and each edge is associated with a name that represents a goal. The first node in the process network must be labeled START. It is also good practice to label final nodes in the network with a name like END or FINISH. But any node that is a sink (has no outgoing edges) is considered to be a final or terminating node.

Each edge in your graph is associated with a goal to be achieved, and the description of this goal is stored in the edge *name.* The syntax of this goal description is as follows:

```
<goal expression> ::= (<edge-sym> <expression>)

<expression> ::= <literal>
             ::= <compound expression>

<compound expression> ::= (& <expression> ... <expression>)
                      ::= (V <expression> ... <expression>)

<edge-sym> ::= <goal-sym>
           ::= =>
```

where <literal> and <goal-sym> are as defined above. For example, an edge might be associated with a name of form:

```
(? (V (P $x @y) (~ (Q a b $x))))
```

Intuitively, the edge symbols !, ?, =>, * are each used to denote a different kind of goal. The * symbol is used to denote a goal that is satisfied by any sequence of states that represents performance of the prescribed expression. For example

```
(* (MOVE loc1 loc2))
```

would be satisfied by any sequence of states over which an object moves from location loc1 to location loc2. The ! symbol is used to denote the *achievement* of the expression, i.e., it is satisfied by a sequence of states whose last state satisfies the expression. For example,

```
(! (LOCKED door1))
```

would be satisfied by any sequence of states in which the door is locked in the last state. In contrast, the ? symbol is used for denoting a goal of testing the truth of the given expression. For example,

```
(? (ACIDIC blob))
```

would be satisfied by any sequence of states in which it is determined that blob was acidic in the first state. Finally, => is used to denote the goal of placing the given expression as a fact in the data base. It is thus satisfied by a sequence of states if the data base contains the given expression in the last state of the sequence.[11]

When your processes file is loaded into the PRS system (using the LOAD menu), your process descriptions are "compiled." What this does is convert the information represented by a process network into a lisp program which embodies the semantics of that network. Each time the process is invoked, this program is executed. To make sure you have set up a process description correctly, you may want to try compiling it before you actually load it later. To do this, type

```
(compile-ka 'SPACENAME)
```

to the GRASPER Listener window. To compile *all* of the spaces in the current GRASPER graph, type

```
(compile-all-kas)
```

to the GRASPER Listener. If you want to see the lisp code associated with your process, type

```
(GET 'SPACENAME 'INTERPRETED-FORM)
```

Also, you can pretty-print any list by using the command

```
(grind-top-level (..the list..))
```

Finally, before you leave GRASPER, it is good practice to save your process descriptions in the file referred to above as YOUR-FILESERVER:>USERNAME>YOUR-APPLICATION-PROCESSES.lisp (click GRAPH and then OUTPUT, and then type the name of your processes file).

[11]For further information about the actual semantics of a process, please refer to the IJCAI-85 paper entitled "A PROCEDURAL LOGIC" by Georgeff, Lansky, and Bessiere.

4.1 Meta-Level Processes, Actions

As we have mentioned above, every PRS system includes as set of default system processes which are loaded when the entire system is loaded. The file containing these default processes is `r:>pes>demo>default-processes.lisp`, and the processes that are found in this file are described in detail in appendix D. Most of the default processes are *meta-level* processes – i.e. they are processes which operate on goals and facts that pertain to the system itself, rather than just the application domain. The file also includes "pseudo" processes called *actions*.

Actions are simply processes with no body – i.e. they are not associated with a network of nodes and edges. Whenever an action responds to a goal, the interpreter executes a specified a lisp function (rather than executing a process network as is normally done). For example, one action in `default-processes.lisp` is `<`. It can be invoked by a goal of form `(! (< $x $y))` or `(? (< $x $y))` and its invocation results in execution of the normal lisp function `<`, applied to the two parameters `$x` and `$y`.

If you have written a lisp function which you would like executed in order to achieve a goal, or if you wish a standard lisp function to be used to satisfy a goal, you may create an "action process" in your processes file, as follows.

Like before, create a new space by first clicking **SPACE** and **CREATE**, and then supply the name of the action. The value of the space, however, must be of a different form than for a normal process definition. The syntax is as follows:

```
<value of an action space> ::=
((ACTION <lisp function name>) (INVOCATION-PART <invocation condition>))
```

where `<invocation condition>` is as defined before. After you have completed this step, you are done. No nodes or edges should be placed in the space describing your action.

5 Database File: YOUR-APPLICATION-DATA-BASE.lisp

This file contains all the facts that you wish loaded initially into the system database. Each fact should be in lisp predicate form, for example: (P x y z). When your database file is loaded, each fact in the file will be *asserted*. The **assert** command inserts facts into the database *without* checking to see if they make any new processes relevant. In contrast, the **conclude** command, in addition to adding the fact to the database, also checks for new relevant processes. This command is described in the next section.

6 User Functions: YOUR-APPLICATION-FUNCTIONS.lisp

This file should contain any functions which you wish to use as *actions* (see section 4.1), or functions that you wish to use to make running your system easier. For example, I use the file **r:>pes>rcs>rcs-functions.lisp** to define functions which start up the RCS application system by concluding certain key facts into the database. The conclude command not only adds these facts to the database, but also checks to see if any new processes are relevant and if so, invokes them.

A conclude operation may be utilized within a user-defined functions by including a command of the following form:

```
(send *system-data-base* :conclude '(...fact...) nil)
```

Note that a process edge labeled by a goal expression of form **(=> <expression>)** also represents the act of *concluding* that expression into the database. It thus can result in the execution of new relevant processes.

Another way to start up an application system is simply to push a goal on the system goal stack. If you want to define a function that pushes specific goals on to the goal stack, use the following command:

```
(send *system-goal-stack* :push '(... goal ...) nil nil)
```

Of course, the **RUN** menu can also be used to conclude facts and push goals on the goal stack – the option of utilizing user-defined functions is provided merely as a convenience.

7 Running A System: The RCS system

Once you have set up all your files correctly, running your system is a fairly simple process. I will now go through the list of things one might do to run an already set up system, from login, onward. These instructions may thus be used by those of you who wish to run a system that has already been set up. I will describe the process of running a system by actually leading you through running the RCS (Reaction Control System) application, which performs fault diagnosis for a subsystem of the space shuttle. New users of the system might want to follow these steps as a trial usage of the system.

1. **Login.** Type `(login 'USERNAME)`.

2. **Load the PRS system.** Type `(make-system "pes")`. Answer yes to any questions that might be asked. This may take a few minutes. Then type `SELECT-A` to transfer to the PRS window.

3. **Load the application.** To get the top level PRS menu presented to you, click on any mouse button. Then click `LOAD`. We will begin by loading the processes file for the RCS system. Click `INITIALIZE PROCESSES`, then click `FILE`, and type in the file name **`r:>pes>rcs>rcs-processes.lisp`**. When the processes have all be loaded and compiled, you are given the loading menu again. Now click `INITIALIZE DATABASE`, then click `FILE`, and type the file name **`r:>pes>rcs>rcs-database.lisp`**. When database loading is done, you will be given the loading menu again. Finally click `LISP FUNCTIONS` and type the file name **`r:>pes>rcs>rcs-functions.lisp`**.

4. **Viewing processes.** At this point, you may want to look at you application processes. This can be done by clicking `EDIT` in the top level menu, then `CURRENT PROCESSES`. You will then be placed into GRASPER, and can look at any process description you like, by clicking SPACE, then SELECT, and then clicking the desired process name on the menu presented.

5. **Set up your window for running the system.** You can run your system in any PRS window. If you do not care whether you are able to view the RCS processes while you execute them, just return to the PRS window by typing `SELECT-A` and go on to the next step.

   The other alternative is to scale down the PRS window and place it on top of the GRASPER window, with enough room to view the processes. (We are currently working on a nice execution monitoring system that will do this automatically for you, and even highlight process edges as they are traversed.) To get this all set up right, proceed as follows. Go back to the PRS window (type `SELECT-A`). Do a DOUBLE-CLICK on the right button, then click `EDIT SCREEN` followed by

RESHAPE. You will then be given a circular icon which you should place over the PRS window and click (this indicates which window you are going to reshape). Then you will be given a window corner. Place it where you would like the upper left-hand corner of the PRS window to be, and rubber band it to the size window you like, and finish by clicking the button. Then click EXIT in the EDIT SCREEN menu.

If you later wish to expand this window to its normal size, once again do a DOUBLE-CLICK on the right button, click EDIT SCREEN followed by EXPAND WINDOW, and then place the circular icon over the PRS window and click. Follow this once again by a click on EXIT in the EDIT SCREEN menu.

6. **Run the system**. You have two choices of how to get the system running – through the RUN menu or by invoking an already defined user function. We present the menu system method first.

   Click RUN in the top level PRS menu. You must now conclude four facts to get the RCS system going. This is because the main RCS procedure which performs diagnosis for this subsystem of the space shuttle is activated by various alarms in the shuttle. Each of the four facts you conclude to the system represents an alarm that has gone off. Because the invocation part of the top level diagnosis procedure matches on these facts, it will start executing after the fourth fact is added. The processes that respond to the first three facts (as well as the fourth) are meta-level system processes that perform data base update and maintenance.

   Proceed now by entering the four facts listed below. For each fact, click FACT on the RUN menu, and type in the fact.

```
(LIGHT RCS-JET)
(ALARM BACKUP-CW)
(FAULT RCS.1 RCS THR.1.1 JET)
(JETFAIL-INDICATOR ON MIV.1.1.1)
```

   After you have concluded the last fact, the main diagnosis process, JET-FAIL-ON should get fired up. Lots of textual tracing will be presented to you. (We are currently working on a tracing facility that is more graphical in nature.) At some points, questions will be asked. Type YES or NO, as asked. If you are asked for the pressure of some meter, just type an integer number. (Something between 0 and 500 is appropriate.) You may also be asked to type in a particular message or string of your choice, in quotes. Eventually, the diagnosis will finish and the RUN menu will be presented again.

   If you want, you can experiment with the tracing mechanism by clicking TRACE in the top level menu, and then clicking NO-TRACING or TRACE-SOME-PROCESSES and providing a subset of processes to trace. Then try running the system again.

(Unless you reinitialize the database totally again, you need to conclude only one of the four facts given above to get the system fired up again – the facts were never deleted from the previous round.)

As mentioned above, an alternate way of getting the system going is to use a user-defined function. Type to the PRS window: `(RCSINIT 'RCS.1 'THR.1.1 'MIV.1.1.1)`. This function is defined in `r:>pes>rcs>rcs-functions.lisp` and loads all four of the facts described above. To load only one fact, you can use the function `(REPROMPT)`. To delete all four facts, type `(RCSDELETE 'RCS.1 'THR.1.1 'MIV.1.1.1)`.

7. **Logout.** To logout from the lisp machine, type `(logout)` to any PRS or Lisp Listener window.

A Running the Robot Simulator

The SRI Robot was designed by Stan Reifel and the graphical simulator by Leslie Pack Kaelbling. To run the PRS robot application system, you must also have the robot simulator loaded. The whole process of setting up both systems is as follows:

1. Go to the Lisp Listener window (type `SELECT-L`). Make sure you are in the user package by typing `(PKG-GOTO 'USER)`.
2. Load the simulator: `(make-system "robot-sim")`. Type yes to any questions you may be asked.
3. Type `SELECT-R` to get to the simulator. You will be given a corner of the simulator window. Rubber-band it to the size you want, but be sure that you fill at most 2/3 of the screen.
4. Click READ FILE and load Leslie's maze which is in `r:>pack>rex>maze2.lisp`.
5. Click CHANGE WORLDS and type `maze2`.
6. Scale the maze to fit in the window. (Click `SCALE` and read the instructions.)
7. Make sure the attributes of the robot simulator window are set to "let output happen" (Click double right on the mouse, then click `ATTRIBUTES`, then click on `Let it happen` and then `Do it`).
8. Click START SIMULATION, and when asked for a user function type: `(SET-VELOCITY 20 20)`. Your command should be echoed in the command monitoring window of the simulator. If it is not, repeat this command.
9. If you have not done so already, now load the PRS system. Go back to the Lisp Listener window (`SELECT-L`) and type `(make-system "pes")`. Answer yes to any questions that might be asked. Finally, type `SELECT-A` to go to the PRS window.
10. Once your are in the PRS window, make sure the current package indicator in the bottom banner reads `PES:`. If it does not, type `(PKG-GOTO 'PES)`.
11. Now reshape the PRS window to fit in the space left over by the robot simulator window. *They should not overlap.* Do a DOUBLE-CLICK on the right button, then click `EDIT SCREEN` followed by `RESHAPE`. You will then be given a circular icon which you should place over the PRS window and click (this indicates which window you are going to reshape). Then you will be given a window corner. Place it where you would like the upper left-hand corner of the PRS window to be, and rubber band it to the size window you like, and finish by clicking the button. Then click `EXIT` in the `EDIT SCREEN` menu.

12. In the PRS window type (SETQ USER:*DEBUGGING* NIL).

13. Now load the robot system. To get the top level PRS menu presented to you, click on any mouse button. Then click LOAD. Click INITIALIZE PROCESSES, then click FILE, and type in the file name r:>pes>robot>robot-processes.lisp. When the processes have all be loaded and compiled, you are given the loading menu again. Now click INITIALIZE DATABASE, then click NULL DATABASE (we start with an empty data base for this application). Then you will be given the loading menu again. Finally click LISP FUNCTIONS and type the file name r:>pes>robot>robot-functions.lisp.

14. Get the system running by clicking on RUN in the top level menu. Then click FACT and type in the fact (MAZEREADY).

That should do it! You can stop the simulation by typing control-ABORT in the PRS window. You can also play with turning tracing on and off by using the TRACE menu.

B Lisp Machine Packages

Lisp Machine packages can be somewhat confusing, so a brief word is in order. Packages are the way name-spaces are partitioned on the Lisp Machine. At any point in time, the machine is in a specific package. All tokens that you type to a window are considered prefixed by the current package name. At any point in time, you can see what package you are in by looking at the bottom banner of the window. The default package that is usually in force is USER. To switch to a different package, use the command

```
(PKG-GOTO '<package name>)
```

When any lisp file is loaded, it is loaded into some specific package. The package into which a file is loaded is determined by the header banner of that file. All of the PRS system files are loaded into the PES package. What this means is that *every token* in the PRS system files is considered to be prefixed by the string PES:. This is true for function and variable names, as well as any other token, *including* quoted atoms.

What are the consequences of all of this? *You must be in the PES package when you use this system.* If you are, things should work fine. If you aren't, things will probably go awry!!

C Variable Usage

The design of the PRS system has attempted to stress the use of logical variables – i.e. variables as they are used in PROLOG. Such variables can never be rebound. Unfortunately, in writing PRS processes we have found the need to use variables with other kinds of semantics – in particular, variables that have a semantics similar to what is found in standard programming languages. As a compromise, we have come up with three kinds of variables in the PRS system, all of whose semantics can be mapped onto a standard logical variable semantics, but whose usage differs. *Global variables* are just like logical variables in the classic sense. *Local variables* are like global variables, but have a limited "extent" or "lifetime." *Program variables* function much like normal variables in standard programming languages.

In general, one should use global variables, wherever possible. They are prefixed by `$` and may have only one binding during the lifetime of a particular process instance. Note that each instantiation of a process is associated with its own global variables. Thus, the global variables in each recursive call of the same process will be distinct.

Also note that for any kind of variable V, if the parameter of a goal expression on some edge is bound to V, V will be bound to the corresponding formal variable in the invocation part of the process that is invoked to achieve that goal (i.e. goal parameters are passed via pass-by-reference rather than pass-by-value).

If you must rebind a variable over the course of a process execution (for example, in a loop), you have the choice of using `%` variables (*local variables*) or `@` variables (*program variables*). `@` variables may be rebound from arc to arc and retain their value between arcs. Whether or not the value of an `@` variable is rebound or not will depend upon context. A goal of form `(! (= @x @y))` will, by default, bind the value of `@x` to be equal to the current value of `@y` if it has one. If `@y` was *not* bound and `@x` was bound when this goal occurred, `@y` would be bound to the value of `@x`. If both were unbound, they would nevertheless be "bound together," and if one achieved a binding later, the other would be bound to that binding as well. A goal of form `(? (= @x @y))` simply tests to see if the bindings of the two variables are equal.

In contrast to `@` variables, `%` variables are more like global variables, except their binding is meaningful only on a per-edge basis. In other words, on each edge, the appearance of a `%` variable is similar to the creation of a fresh new global variable. The binding of a `%` variable will *not* carryover from one edge to the next. However, if multiple goals are invoked on the same edge, the same variable and binding will be used for all of them.

Clearly, both global and local variables are *logical* variables in the classic sense – they cannot be rebound. In order to handle the semantics of `@` variables consistently within this logical-variable framework, the binding of program variables may be viewed as a particular "aspect" or slot value of an associated global variable. Whereas global variables can be bound only once, it is natural to think of the value of its slot as being rebindable. For example, one could imagine a global variable as a box. Each global variable is a specific box and can never be rebound to another box, but different objects may be placed *in* the box at different points in time.

D Default System Processes

The processes described below are a default part of every PRS system and are located in the file `r:>pes>demo>default-processes.lisp`. They may be subdivided into two categories, meta-level processes, and actions. Moreover, some of these actions are actually *special actions* – i.e. they are meta-level actions whose associated functions can manipulate the internals of the PRS system.

The meta-level processes are as follows:

- `s-basic`
  This process helps choose which process to execute from several that match a given goal. It has an invocation part of form:

```
(AND (*GOAL (? (HOW-TO-REACH $GOAL $RELEVANT-KAS)))
     (*LISP-PREDICATE (NOT (FACTS-AMONG? $RELEVANT-KAS))))
```

- `s-select-fact`
  If both facts and processes match a goal, this process makes sure that facts are used first. It has an invocation part of form:

```
(AND (*GOAL (? (HOW-TO-REACH $GOAL $RELEVANT-KAS)))
     (*LISP-PREDICATE (FACTS-AMONG? $RELEVANT-KAS)))
```

- `s-data-base-management1`
  A process that handles data base management. It is used when a fact is concluded, but no new processes are relevant nor are there any goals it achieves. It has an invocation part of form:

```
(AND (*GOAL (? (HOW-TO-USE $STATEMENT
                           $ACHIEVED-GOALS $NEW-RELEVANT-KAS)))
     (*LISP-PREDICATE (NULL $ACHIEVED-GOALS))
     (*LISP-PREDICATE (NULL $NEW-RELEVANT-KAS)))
```

- `s-data-base-management2`
  A process that handles data base management. It is used when a fact is concluded and there are goals it achieves. It has an invocation part of form:

```
(AND (*GOAL (? (HOW-TO-USE $STATEMENT
                           $ACHIEVED-GOALS $NEW-RELEVANT-KAS)))
     (*LISP-PREDICATE (NOT (NULL $ACHIEVED-GOALS)))
```

- `s-data-base-management3`
  A process that handles data base management. It is used when a fact is concluded and it doesn't achieve any goals, but there are new relevant processes. It has an invocation part of form:

```
(AND (*GOAL (? (HOW-TO-USE $STATEMENT
                           $ACHIEVED-GOALS $NEW-RELEVANT-KAS)))
     (*LISP-PREDICATE (NULL $ACHIEVED-GOALS))
     (*LISP-PREDICATE (NOT (NULL $NEW-RELEVANT-KAS)))
     (*LISP-PREDICATE
        (EQUAL-TOP-OF-STACK?
          (LIST '? (LIST 'HOW-TO-USE $STATEMENT
                                     $ACHIEVED-GOALS
                                     $NEW-RELEVANT-KAS)))))
```

- clean-up-mechanism
  A process that does data base management. In particular, it is intended to invoke certain truth maintenance functions, including updating of time-sensitive data. It has an invocation part of form:

  ```
  (*GOAL (! (CLEAN-UP $STATEMENT)))
  ```

- s-select-ka-1
  If a new fact makes exactly one process relevant, this process invokes it. It has an invocation part of form:

  ```
  (AND (*GOAL (! (SELECT-KA $N-RELEVANT-KAS)))
       (*LISP-PREDICATE (EQUAL 1 (LENGTH $N-RELEVANT-KAS))))
  ```

- user-question
  This process asks the user whether a given fact is true or not, and inserts the truth or falsity of that fact into the database. Also, if the user responds "WHY," an action will be invoked that places the user into the Lisp Machine data inspector. This process has an invocation part of form:

  ```
  (*GOAL (! (USER-QUESTION $FACT)))
  ```

The actions within default-processes.lisp are given below. For each action we give the name of the lisp function with which it is associated, and its invocation part.

- process-ka
  Lisp function: *process-ka* (a function in the intepreter)
  Invocation part: (*GOAL (! (PROCESS-KA $KA-NAME $GOAL-EXPRESSION)))

- process-ka-fact
  Lisp function: *process-ka-fact* (a function in the intepreter)
  Invocation part: (*GOAL (! (PROCESS-KA-FACT $KA-NAME $ENVIRONMENT $ID)))

- `process-achieved-goals`
  Lisp function: *process-achieved-goals* (a function in the intepreter)
  Invocation part: `(*GOAL (! (PROCESS-ACHIEVED-GOALS $ACHIEVED-GOALS)))`

- `action?`
  Lisp function: *action?* (a function in the intepreter)
  Invocation part: `(*GOAL (! (ACTION? $KA-NAME)))`

- `assert`
  Lisp function: *assert* (a function in the intepreter)
  Invocation part: `(*GOAL (! (ASSERT $FACT)))`

- `print-list`
  Lisp function: *print-list* (a function in the intepreter)
  Invocation part: `(*GOAL (! (PRINT-LIST $LIST)))`

- `why`
  Lisp function: *explanation* (a function in the intepreter)
  Invocation part: `(*FACT (SAID USER WHY))`

- `print`
  Lisp function: *print* (the usual lisp function)
  Invocation part: `(*GOAL (! (PRINT $X)))`

- `listp`
  Lisp function: *listp* (the usual lisp function)
  Invocation part: `(*GOAL (! (LISTP $X)))`

- `atom`
  Lisp function: *atom* (the usual lisp function)
  Invocation part: `(*GOAL (! (ATOM $X)))`

- `null`
  Lisp function: *null* (the usual lisp function)
  Invocation part: `(*GOAL (! (NULL $X)))`

- `<`
  Lisp function: $<$ (the usual lisp function)
  Invocation part: `(OR (*GOAL (! (< $X $Y))) (*GOAL (? (< $X $Y))))`

- `<=`
  Lisp function: $\leq$ (the usual lisp function)
  Invocation part: `(OR (*GOAL (! (<= $X $Y))) (*GOAL (? (<= $X $Y))))`

- \>
  Lisp function: > (the usual lisp function)
  Invocation part: `(OR (*GOAL (!  (> $X $Y))) (*GOAL (?  (> $X $Y))))`

- \>=
  Lisp function: ≥ (the usual lisp function)
  Invocation part: `(OR (*GOAL (!  (>= $X $Y))) (*GOAL (?  (>= $X $Y))))`

The *special actions* (meta-level actions) within `default-processes.lisp` are given below.

- `s-~ -naf`
  This process implements negation as failure for both database facts and goals.
  Lisp function: *special-naf* (a function in the PRS interpreter)
  Invocation part:

```
(AND (*GOAL (? (HOW-TO-REFORMULATE $GOAL)))
     (*LISP-PREDICATE (EQUAL-TOP-OF-STACK?
                          (LIST '? (LIST 'HOW-TO-REFORMULATE $GOAL))))
     (*LISP-PREDICATE (EQUAL '~ (CAADR $GOAL))))
```

- `s-&`
  This process is used to achieve a conjunction of goals.
  Lisp function: *special-and* (a function in the PRS interpreter)
  Invocation part:

```
(AND (*GOAL (? (HOW-TO-REFORMULATE $GOAL)))
     (*LISP-PREDICATE (EQUAL-TOP-OF-STACK?
                          (LIST '? (LIST 'HOW-TO-REFORMULATE $GOAL))))
     (*LISP-PREDICATE (EQUAL '& (CAADR $GOAL))))
```

- `s-V`
  This process is used to achieve a disjunction of goals.
  Lisp function: *special-or* (a function in the PRS interpreter)
  Invocation part:

```
(AND (*GOAL (? (HOW-TO-REFORMULATE $GOAL)))
     (*LISP-PREDICATE (EQUAL-TOP-OF-STACK?
                          (LIST '? (LIST 'HOW-TO-REFORMULATE $GOAL))))
     (*LISP-PREDICATE (EQUAL 'V (CAADR $GOAL))))
```

- `s-=`
  This is a special process for doing equality. The code differentiates between equality for goals of form `!` and ?. In the case of a goal of form `(!  (= ...))`, the code tries to force equality (it is like an assignment statement). In the case

of a goal of form `(? (= ...))`, the code tests for equality.
Lisp function: *special-equality* (a function in the PRS interpreter)
Invocation part:

```
(OR (*GOAL (? (= $X $Y)))
    (*GOAL (! (= $X $Y))))
```

- pushgoal
  This process pushes a goal on the goal stack and sets a parameter to indicate whether or not the goal was successfully achieved. If it was successful, the environment is updated accordingly.
  Lisp function: *special-pushgoal* (a function in the PRS interpreter)
  Invocation part:

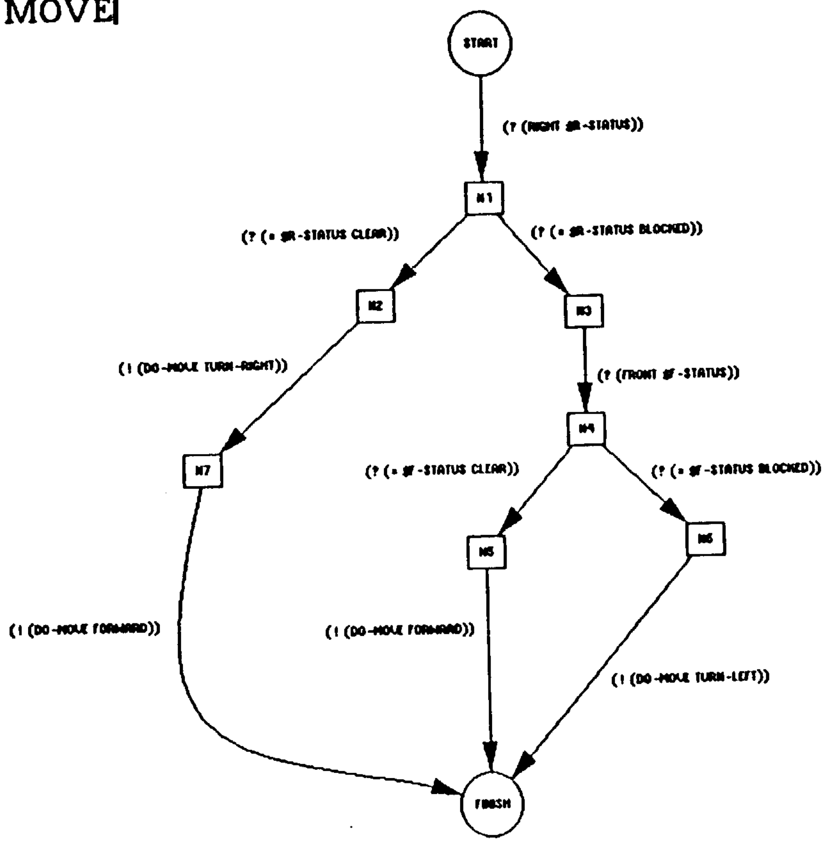
```
(*GOAL (! (PUSHGOAL $GOAL $SUCCEED)))
```

E Sample Processes

On the next few pages we have included a set of sample process descriptions for your perusal. The invocation part associated with each process is given in the GRASPER Listener window.

MAZE-MOVE

GRAPH
SPACE
NODE
EDGE



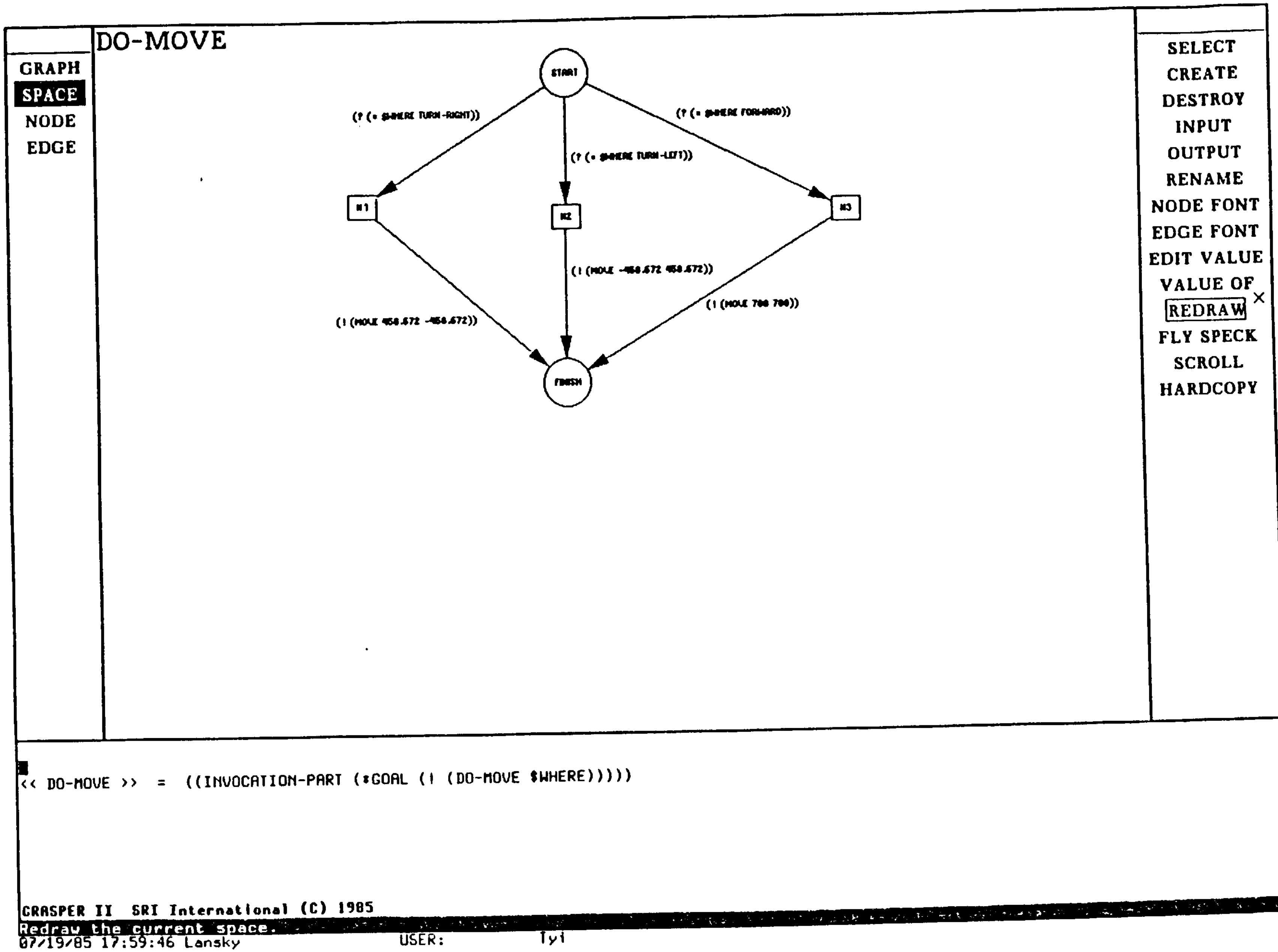
SELECT
CREATE
DESTROY
INPUT
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
SCROLL
HARDCOPY

X

```
<< MAZE-MOVE >>  =  ((INVOCATION-PART (*GOAL (! (MAZE-MOVE)))))
```

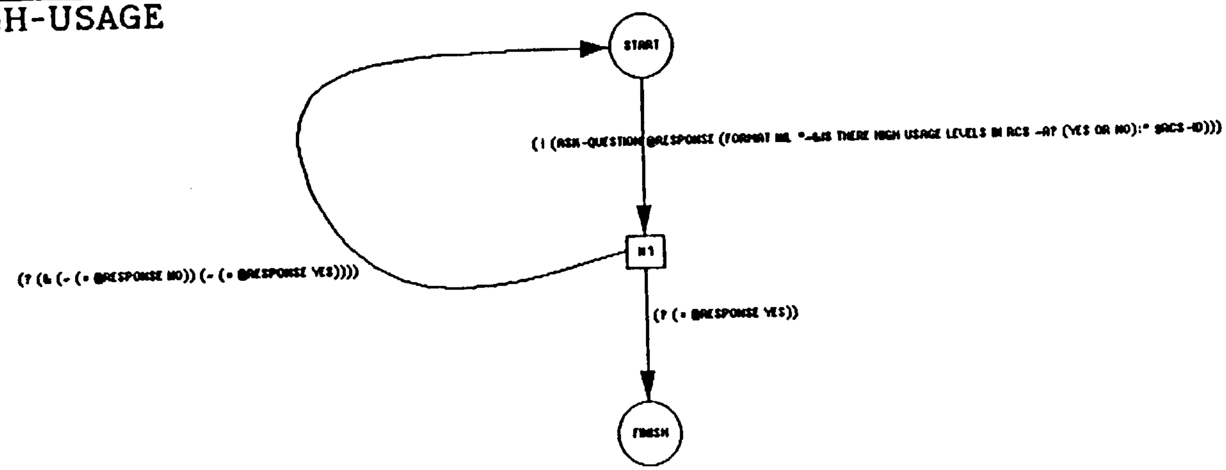
Algorithm for moving a
robot through a maze

GRASPER II SRI International (C) 1985
Make a hardcopy of this space.
07/19/85 17:55:00 Lansky   USER: Tyi


DO-MOVE
GRAPH
SPACE
NODE
EDGE
START
N1
N2
N3
FINISH
(? (= $WHERE TURN-RIGHT))
(? (= $WHERE TURN-LEFT))
(? (= $WHERE FORWARD))
(! (MOVE 458.672 -458.672))
(! (MOVE -458.672 458.672))
(! (MOVE 700 700))
SELECT
CREATE
DESTROY
INPUT
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
SCROLL
HARDCOPY
<< DO-MOVE >> = ((INVOCATION-PART (*GOAL (! (DO-MOVE $WHERE)))))
GRASPER II SRI International (C) 1985
Redraw the current space.
07/19/85 17:59:46 Lansky USER: Tyi

GRAPH
SPACE
NODE
EDGE

HIGH-USAGE

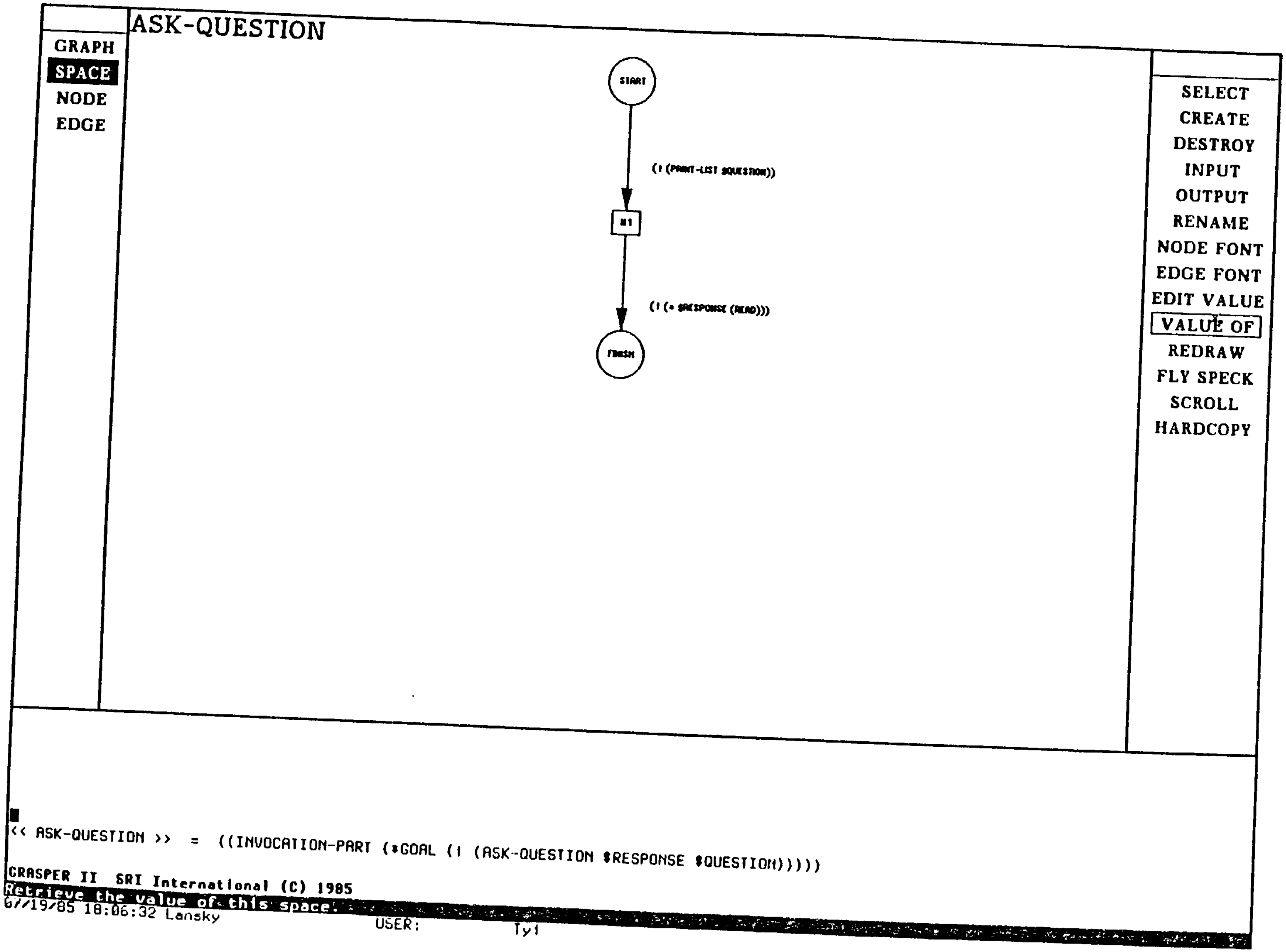


SELECT
CREATE
DESTROY
INPUT
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
SCROLL
HARDCOPY

```
<< HIGH-USAGE >>  =  ((INVOCATION-PART ($GOAL (? (HIGH-USAGE $RCS-ID)))))
```

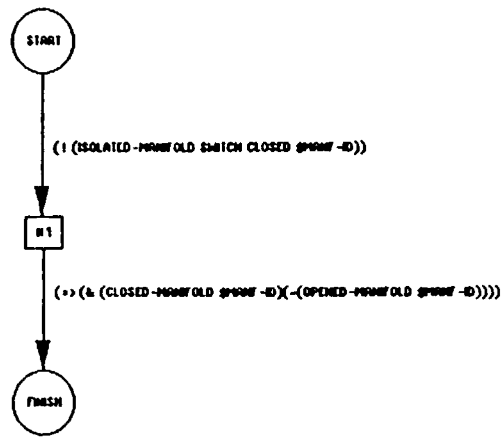
A test for determining if there is High Usage of a particular system.

GRASPER II SRI International (C) 1985
Make a hardcopy of this space.
07/19/85 18:05:16 Lansky   USER: Tyi


GRAPH
SPACE
NODE
EDGE
ASK-QUESTION
START
(! (PRINT-LIST $QUESTION))
N1
(! (= $RESPONSE (READ)))
FINISH
SELECT
CREATE
DESTROY
INPUT
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
SCROLL
HARDCOPY
<< ASK-QUESTION >> = ((INVOCATION-PART (*GOAL (! (ASK-QUESTION $RESPONSE $QUESTION)))))
GRASPER II SRI International (C) 1985
Retrieve the value of this space.
07/19/85 18:06:32 Lansky
USER: Ty1

GRAPH
SPACE
NODE
EDGE

CLOSED-MANIFOLD



SELECT
CREATE
DESTROY
INPUT
OUTPUT
RENAME
NODE FONT
EDGE FONT
EDIT VALUE
VALUE OF
REDRAW
FLY SPECK
SCROLL
HARDCOPY

```
<< CLOSED-MANIFOLD >>  =  ((INVOCATION-PART (AND (*GOAL (! (CLOSED-MANIFOLD $MANF-ID))) (*FACT (TYPE MANF-ISOL-VALVE $N $MANF-ID)) (*LI
SP-PREDICATE (NOT (= $N 5.))))))
```

Achieving a closed manifold (for manifolds not of type 5) and concluding facts about its closure to the database.

GRASPER II SRI International (C) 1985

07/19/85 18:09:18 Lansky   USER: Tyi